

Homomorphic Evaluation of Database Queries

Sudharaka Palamakumbura and Hamid Usefi*

Department of Mathematics and Statistics
Memorial University of Newfoundland
St. John's, NL, Canada, A1C 5S7
{sudharakap, usefi}@mun.ca

Abstract. Homomorphic encryption is an encryption method that enables computing over encrypted data. This has a wide range of real world ramifications such as being able to blindly compute a search result sent to a remote server without revealing its content. This paper discusses how database search queries can be made secure using a homomorphic encryption scheme. We propose a new database search technique that can be used with the ring-based fully homomorphic encryption scheme proposed by Brakerski.

Keywords: homomorphic; privacy; encryption; database; query

1 Introduction

According to a recent study, 74% of smartphone users use a location based service (such as Google Maps) to find directions and other location based information [1]. Moreover, the adaptation of these kind of services in healthcare are becoming increasingly common with cloud-based health recording and genomic data management tools such as Microsoft Health. However, the widespread adaptation of location-based services poses a threat to users because their personal data, such as location, health records, and sometimes even genomic data, is shared on the web without any guarantee of privacy.

In this work, we address the problem of searching privately on a database. We consider the scenario that a user wants to send a search request to a server and would want the server to learn nothing about his query. So, it makes sense that the user encrypts his search using his public key and sends the cipher-text over to the server. We consider the case where data over the server is not encrypted. This applies in particular to queries sent to search engines. The case where the data over the server is encrypted will be treated differently elsewhere. Our proposed scheme shall use a homomorphic encryption scheme.

Homomorphic encryption allows computations to be carried out on the cipher-text such that after decryption, the result would be the same as carrying out identical computations on the plain-text. This has novel implications such as being able to carry out

* The research is supported by NSERC of Canada under grant # RGPIN 418201 and the Research & Development Corporation of Newfoundland and Labrador.

operations on database queries in the form of cipher-text and returning the result to the user so that no information about the query is revealed at the server's end [9].

The idea of homomorphic encryptions is not new, and even the oldest of ciphers, ROT13 developed in ancient Rome, had homomorphic properties with respect to string concatenations [7]. Certain modern ciphers such as RSA and El Gamal also support homomorphic multiplication of cipher texts [7].

The idea of a “fully” homomorphic encryption scheme (or *privacy homomorphism*) which supports two homomorphic operations was first introduced by Rivest, Adleman, and Dertouzos in 1978 [6]. After more than three decades, the first fully homomorphic encryption scheme was founded by Gentry in 2009 with his breakthrough construction of a lattice based cryptosystem that supports both homomorphic additions and multiplications [8]. Although the lattice based system is not used in practice, it paved the way for many other simpler and more efficient fully homomorphic models constructed afterwards.

At a high level, Gentry's idea can be described by the following general model. This is the blueprint that is used in all homomorphic encryption schemes that followed.

1. Develop a *Somewhat Homomorphic Encryption Scheme* that is restricted to evaluating a finite number of additions or multiplications.
2. Modify the somewhat homomorphic encryption scheme to make it *Bootstrappable*, that is, modifying it so that it could evaluate its own decryption circuit plus at least one additional NAND gate.

Every probabilistic encryption function usually introduces a *noise* and when the noise exceeds a certain threshold, the decryption function does not return the desired plain-text. The idea behind constructing a bootstrappable scheme is that whenever the noise level is about to reach the threshold, we can *bootstrap* the cipher-text and get a new cipher-text so that these cipher-texts decrypt to the same plain-text but the new cipher-text will have a lower noise. In this way, if the cipher-text is bootstrapped from time to time, an arbitrary number of operations can be carried out.

Our work improves upon a method proposed by Gahi et al. [3] to homomorphically encrypt database queries. Their work specifically uses the DGHV fully homomorphic encryption scheme [4]. The DGHV scheme operates on plain-text bits separately, and thus Gahi's method requires a large amount of computations to perform even on a simple operation such as integer multiplication. We propose an alternative to Gahi's method, which we call *Homomorphic Query Processing*. Our method is not restricted to the DGHV scheme and can be used with more modern fully homomorphic encryption schemes. For example, using our Homomorphic Query Processing technique with the more recent ring based fully homomorphic encryption scheme proposed by Brakerski et al. [5], which work on blocks of data (such as integers) rather than single bits (as in Gahi's scheme), the number of computations can be greatly reduced.

2 DGHV Fully Homomorphic Encryption

The DGHV scheme was introduced by Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan in 2010, and this scheme operates on integers as opposed to lattices in Gentry's original construction. The scheme follows Gentry's original blueprint by first constructing a somewhat homomorphic encryption scheme. The key generation, encryption and decryption algorithms of the DGHV scheme are given below.

Let $\lambda \in \mathbb{N}$ be the security parameter and set $N = \lambda$, $P = \lambda^2$ and $Q = \lambda^5$. The scheme is based on the following algorithms;

- **KeyGen**(λ): The key generation algorithm that randomly chooses a P -bit integer p as the secret key.
- **Enc**(m, p): The bit $m \in \{0, 1\}$ is encrypted by

$$c = m' + pq,$$

where $m' \equiv m \pmod{2}$ and q, m' are random Q -bit and N -bit numbers, respectively. Note that we can also write the cipher-text as $c = m + 2r + pq$ since $m' = m + 2r$ for some $r \in \mathbb{Z}$.

- **Dec**(c, p): Output $(c \bmod p) \bmod 2$ where $(c \bmod p)$ is the integer c' in $(-p/2, p/2)$ such that p divides $c - c'$.

The value m' is called the *noise* of the cipher-text. Note that this scheme, as it is given above, is symmetric (i.e., it only has a private key). We can define the public key as a random subset sum of encryptions of zeros, that is, the public key is a randomly chosen sum from a predefined set of encryptions of zeros: $S = \{2r_1 + pq_1, 2r_2 + pq_2, \dots, 2r_n + pq_n\}$. A typical encryption of the plain-text m would be,

$$\begin{aligned} c &= m + \sum_{i \in T} (2r_i + pq_i) \\ &= m + 2 \sum_{i \in T} r_i + p \sum_{i \in T} q_i, \end{aligned}$$

where $T \subseteq S$. From here on we shall use m' to denote $m + \sum_{i \in T} r_i$ and q to denote $\sum_{i \in T} q_i$.

This scheme is homomorphic with respect to addition and multiplication and decrypts correctly as long as the noise level does not exceed $p/2$ in absolute value. That is, $|m'| < p/2$. Hence, this is a somewhat homomorphic encryption scheme in the sense that once the noise level exceeds $p/2$, the scheme loses its homomorphic ability. It is shown that this scheme is Bootstrappable.

3 Query Processing Using the DGHV Scheme

The DGHV scheme can be used to create a protocol that establishes blind searching in databases. This method was proposed by Gahi et al. [3].

Suppose we need to retrieve a particular record from the database. Typically, we send a query to the database encrypted using the DGHV scheme. Let v_i be the i^{th} bit of the query v and c_i be the i^{th} bit of a record R in database D . Both the query and the database record is encrypted using the DGHV scheme. Suppose the plain-text bit corresponding to v_i is m_i and the plain-text bit corresponding to c_i is m'_i . Then,

$$v_i = m_i + 2r_i + pq_i$$

and

$$c_i = m'_i + 2r'_i + pq'_i,$$

where r_i, r'_i, q_i and q'_i are random numbers and p is the secret key. The server shall compute the following sum for each record R_t with index t :

$$I_t = \prod_i (1 + c_i + v_i). \quad (1)$$

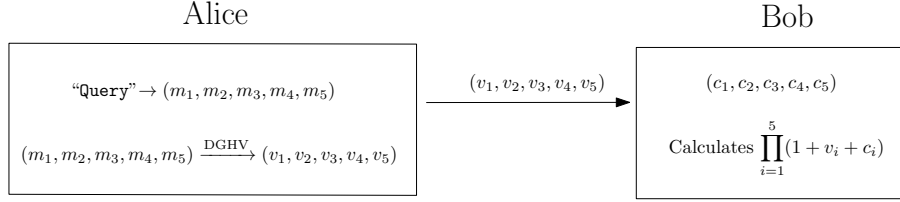


Fig. 1. Calculation of I_r values.

We observe that

$$1 + c_i + v_i = 1 + (m_i + m'_i) + 2(r_i + r'_i) + p(q_i + q'_i).$$

So, if $m_i = m'_i$, then $m_i + m'_i \equiv 0 \pmod{2}$. In this case:

$$1 + c_i + v_i = \text{Enc}(1).$$

On the other hand, if $m_i \neq m'_i$, then $m_i + m'_i \equiv 1 \pmod{2}$. Therefore,

$$\begin{aligned} 1 + c_i + v_i &= 2(1 + r_i + r'_i) + p(q_i + q'_i) \\ &= \text{Enc}(0). \end{aligned}$$

This results in $I_t = \text{Enc}(0)$. Hence, for each record R_t in the database we will have an I_t value that is equal to $\text{Enc}(1)$ or $\text{Enc}(0)$ depending on whether the search query m matches R_t or not.

Next, we calculate the partial sums of the I_t values:

$$S_r = \sum_{t \leq r} I_t. \quad (2)$$

| Database Records | I_r | S_r |
|------------------|--------|--------|
| (1, 1, 0, 0) | Enc(1) | Enc(1) |
| (1, 0, 1, 0) | Enc(0) | Enc(1) |
| (1, 1, 0, 0) | Enc(1) | Enc(2) |
| (1, 1, 0, 1) | Enc(0) | Enc(2) |
| (1, 0, 0, 0) | Enc(0) | Enc(2) |

Table 1. Sample database with corresponding I_r and S_r values

As an example, let us consider a database that has five records, each encoded with 4 bits. If the query sent by the user is (Enc(1), Enc(1), Enc(0), Enc(0)), we obtain the corresponding I_r and S_r values, as shown in Table 1.

Next, we calculate the sequence $(I'_{r,j})$ for every record R_r with index r and every positive integer $j \leq r$:

$$I'_{r,j} = I_r \prod_i (1 + \bar{j}_i + S_{r,i}), \quad (3)$$

where $S_{r,i}$ is the i^{th} bit of S_r and \bar{j}_i represents the i^{th} bit of the encryption of j . Hence, these sequences have the property that whenever $I_r = \text{Enc}(1)$ and $S_r = \text{Enc}(j)$, we have $I'_{r,j} = \text{Enc}(1)$. Otherwise, $I'_{r,j} = \text{Enc}(0)$. Following the example given in Table 1, we get

$$\begin{aligned} (I'_1) &= (\text{Enc}(1)), \\ (I'_2) &= (\text{Enc}(0), \text{Enc}(0)), \\ (I'_3) &= (\text{Enc}(0), \text{Enc}(1), \text{Enc}(0)), \\ (I'_4) &= (\text{Enc}(0), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0)), \\ (I'_5) &= (\text{Enc}(0), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0)). \end{aligned}$$

Finally, we calculate,

$$(R') = \sum_k \text{Enc}(R_k)(I'_k), \quad (4)$$

where R_k is the k^{th} record in D . So, (R') is a sequence containing only the encrypted records that matches our search query. Note that the definition of (R') relies on adding vectors of different lengths. This is done in the natural way, whereby all the vectors are made the same length by padding with zeros prior to addition. In the above example, we obtain,

$$(R') = (\text{Enc}(R_1), \text{Enc}(R_3), \text{Enc}(0), \text{Enc}(0)).$$

At this point, the sequence (R') will contain all the records that match our query, but with trailing encryptions of zeros we do not need. Hence, a second sum is calculated at the server side to determine the number of terms that are useful in the sequence:

$$n = \sum_r I_r$$

This result can be returned to the user and decrypted to obtain the number of records that match the search query. Hence, the sequence (R') can be truncated at the appropriate point and returned to the user for decryption. The whole process is illustrated in Figure 2.

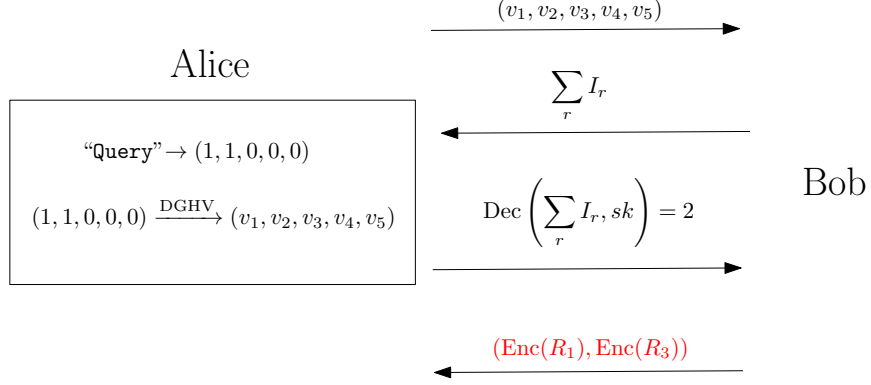


Fig. 2. Alice, Bob, and Gahi's Protocol.

An update query can be performed by,

$$R_{new} = (1 + I_r)R + I_r U, \text{ for every } R \in D,$$

where U is the new value that we wish to insert whenever the query matches R (or $I_r = \text{Enc}(1)$). A deletion of a record can be performed by,

$$R_{new} = (1 + I_r)R \text{ for every } R \in D.$$

To perform all these operations without exceeding the maximum noise permitted $(p/2)$, it is necessary to choose the parameters N , P , and Q appropriately.

Gahi's method works on plain-text bits and thus requires significant computational ability on the part of the server. This is due to the fact that it is restricted to the DGHV scheme which processes plain-text bits separately. Now we propose an alternative protocol called the Homomorphic Query Processing Scheme. This protocol enables us to process database queries using more modern fully homomorphic encryption schemes such as the ring based scheme proposed by Brakerski et al. [5], which acts on blocks of plain-text rather than single bits.

4 Homomorphic Query Processing

The main drawback in Gahi's method is that it requires an enormous number of homomorphic operations because it employs the DGHV encryption scheme, which uses

bitwise encryption. We propose an alternative protocol called *Homomorphic Query Processing* that is compatible with the more recent ring-based fully homomorphic encryption scheme introduced by Braserski et al. [5]. The major advantage is that Braserski's method works on plain-text and cipher-text blocks and thus the number of homomorphic operations required can be greatly reduced.

We first give a brief introduction to the ring based fully homomorphic Encryption Scheme proposed by Braserski, and then proceed to define our Homomorphic Query Processing method.

4.1 Ring Based Fully Homomorphic Encryption

This encryption scheme was introduced by Braserski, et al [5] and operates on the polynomial ring $R = \mathbb{Z}[X]/\langle f(x) \rangle$; the ring of polynomials with integer coefficients modulo $f(x)$, where,

$$f(x) = \prod_{\substack{1 \leq k \leq n \\ \gcd(k, n)=1}} \left(x - e^{2i\pi \frac{k}{n}} \right),$$

is the n^{th} cyclomatic polynomial. The plain-text space is the ring $R_t = \mathbb{Z}_t[x]/\langle f(x) \rangle$, where t is an integer. The key generation and encryption functions make use of two distributions χ_{key} and χ_{err} on R for generating small elements. The uniform distribution χ_{key} is used in the key generation, and the discrete Gaussian distribution χ_{err} is used to sample small noise polynomials. Specific details can be found in [10] and [5]. The scheme is based on the following algorithms.

- **KeyGen**($n, q, t, \chi_{key}, \chi_{err}$): Operating on the input degree n and moduli q and t , this algorithm generates the public and private keys $(pk, sk) = (h, f)$, where $f = [tf' + 1]_q$ and $h = [tgf^{-1}]_q$. Here, the key generation algorithm samples small polynomials from the key distribution $f', g \rightarrow \chi_{key}$ such that f is invertible modulo q and $[\cdot]_q$ denotes coefficients of polynomials in R reduced by modulo q .
- **Encrypt**(h, m): Given a message $m \in R$, the Encrypt algorithm samples small error polynomials $s, e \rightarrow \chi_{err}$ and outputs, $c = [\lfloor q/t \rfloor [m]_t + e + hs]_q \in R$, where $\lfloor \cdot \rfloor$ denotes the floor function.
- **Decrypt**(f, c): Given a cipher-text c , this algorithm outputs, $m = \left[\left[\frac{t}{q} [fc]_q \right] \right]_t \in R$.
- **Add**(c_1, c_2): Given two cipher-texts c_1 and c_2 , this algorithm outputs $c_{add}(c_1, c_2) = [c_1 + c_2]_q$.
- **Mult**(c_1, c_2): Multiplication of cipher-texts is performed in two steps. First, compute $\tilde{c}_{mult} = \left[\left[\frac{t}{q} c_1 c_2 \right] \right]_q$. However, this result cannot be decrypted to the original plain-text using the decryption key f . Therefore, a process known as key switching is done to transform the cipher-text so that it can be decrypted with the original secret key. For more details, we refer to [10].

This encryption scheme is homomorphic with respect to addition and multiplication of plain-texts modulo t . The main advantage in using Braserski's encryption scheme is

that it can be used to encrypt blocks of plain-text instead of dealing with single bits, as in the DGHV scheme [4]. For example, consider the block of plain-text bits, 10100. The integer representation of this block is the value 20. We can represent this integer using the polynomial $X^2 + X^4 = \sum_{i=0}^4 2^i z_i$, where z_i is the i^{th} bit of 10100. In general, if z is an integer and its binary representation is, $z = (\pm 1) \sum_{i=0}^l 2^i z_i$, where $z_i \in \{0, 1\}$ and $l = \lceil \log_2 |z| \rceil$, then we can encode the number z as $\sum_{i=0}^l z_i X^i \in R$.

4.2 Converting the plain-text space into a Field

As we shall see, in our *Homomorphic Query Processing* method, we invert certain plain-text elements and thus the plain-text space should be a field. Therefore, we now discuss how to convert the plain-text ring in Braserski's method to a field. Note that the plain-text space in Braserski's method is defined on the polynomial ring, $R_t = \mathbb{Z}_t[x] / \langle f(x) \rangle$. We shall select $t = p$, where p is a prime number. Then R_p is a field if and only if f is irreducible over \mathbb{Z}_p . Recall that f is the n^{th} cyclomatic polynomial defined as follows:

$$f(x) = \prod_{\substack{1 \leq k \leq n \\ \gcd(k, n) = 1}} \left(x - e^{2i\pi \frac{k}{n}} \right)$$

Let $f(x) = (x - \alpha_1)(x - \alpha_2) \dots (x - \alpha_n)$ be a polynomial defined on $\mathbb{Q}[x]$. The discriminant of f , denoted by $\Delta(f)$, is defined [11] as,

$$\Delta(f) = \prod_{i < j} (\alpha_i - \alpha_j)^2$$

It has been proved in [11] that the n^{th} cyclotomic polynomial reduces modulo all primes if and only if the discriminant of the n^{th} cyclotomic polynomial is a square in \mathbb{Z} . Hence, by choosing a cyclotomic polynomial whose discriminant is not a square we can find a prime p such that f is irreducible over \mathbb{Z}_p . Furthermore, it is shown in [11] that whenever the discriminant of a cyclotomic polynomial f is not a square in \mathbb{Z} , there exist infinitely many primes such that f is irreducible over \mathbb{Z}_p . Thus, we can choose a cyclotomic polynomial with non-square discriminant and check for irreducibility using a standard polynomial irreducibility test such as Rabin's test, until we obtain a prime for which the cyclotomic polynomial is irreducible. For example, even if we consider a large cyclotomic polynomial with non-square discriminant like the 107^{th} cyclotomic (which has degree 106), and consider the primes less than 100, it can be seen that it is irreducible over many primes: $\mathbb{Z}_2, \mathbb{Z}_5, \mathbb{Z}_7, \mathbb{Z}_{17}, \mathbb{Z}_{31}, \mathbb{Z}_{43}, \mathbb{Z}_{59}, \mathbb{Z}_{67}, \mathbb{Z}_{71}, \mathbb{Z}_{73}$ and \mathbb{Z}_{97} .

We now propose our Homomorphic Query Processing scheme, which is compatible with the Braserski's ring based fully homomorphic encryption scheme mentioned previously.

4.3 Homomorphic Query Processing

We begin by defining the value F_i for the i^{th} record (denoted by R_i) in the database. We write $\text{Enc}(m)$ for the $\text{Enc}(m, pk)$, where pk is the public key of the user. Then the

user sends $\text{Enc}(m)$ to the server to search for the records that match m . Now, the server computes the following:

$$F_i = \left(\prod \text{Enc}(m - R_k) \right) \left(\prod \text{Enc}(R_i - R_k)^{-1} \right), \quad (5)$$

where each of the products above is over all the records R_k such that $R_k \neq R_i$. Since we are dealing with a fully homomorphic encryption scheme, we can compute $\text{Enc}(m - R_k)$ values by computing $\text{Enc}(m) - \text{Enc}(R_k)$. Also, since all the R_i values are known to the server, the term $\prod_{R_k \neq R_i} \text{Enc}(R_i - R_k)^{-1}$ can be reduced to a simpler form using the homomorphic property of the encryption scheme in order to perform a single encryption. That idea of defining the F_i 's in this way was inspired by Lagrange Interpolating Polynomials. Indeed, we have

$$\begin{aligned} F_i &= \text{Enc} \left(\prod_{R_k \neq R_i} (m - R_k) \right) \text{Enc} \left(\prod_{R_k \neq R_i} (R_i - R_k) \right)^{-1} \\ &= \text{Enc} \left(\prod_{R_k \neq R_i} \frac{m - R_k}{R_i - R_k} \right) \end{aligned}$$

We remark that whenever $m = R_i$ (query being equal to the record we are comparing), we have $F_i = \text{Enc}(1)$ and $F_i = \text{Enc}(0)$, otherwise. Note that here we are assuming that the query is contained somewhere in the database. If the query is not contained anywhere in the database, an encryption of something other than 1 or 0 will be the output. This special scenario is discussed later.

Now, we define the partial sums of the F_i values as follows:

$$G_i = \sum_{j \leq i} F_j. \quad (6)$$

Using these partial sums, we can then calculate the sequence $(F'_{i,k})$ corresponding to each record as follows,

$$F'_{i,k} = F_i \left(\prod_{j \neq k} G_i - \text{Enc}(j) \right) \left(\text{Enc} \prod_{j \neq k} (k - j)^{-1} \right), \quad (7)$$

where $1 \leq k \leq i$. It can be seen that $F'_{i,k} = \text{Enc}(1)$ if $F_i = \text{Enc}(1)$ and $G_i = \text{Enc}(k)$ are both satisfied. Hence, the sequences $(F'_{i,k})$ have the property that whenever $F_i = \text{Enc}(1)$ (i.e., the i^{th} record matches the query), we have an $\text{Enc}(1)$ at the k^{th} position of the sequence where $G_i = \text{Enc}(k)$. All other entries of the sequence are encryptions of zero. Therefore,

$$(R') = \sum_k \text{Enc}(R_k)(F'_k), \quad (8)$$

where R_k is the k -th record in D will give us a sequence containing only the encrypted records that match our search query. Note that the definition of (R') relies on adding

vectors of different lengths. This is done in the natural way, whereby all the vectors are made the same length by padding with zeros prior to addition.

To further illustrate our scheme, let us consider an example where the database contains five records, each with 4 bits of data. Also, let our encryption scheme encrypt 2 bits at a time. Then, if the search query is $(\text{Enc}(2), \text{Enc}(3))$, the corresponding F_i and G_i values are given in Table 2.

Table 2. Sample database and corresponding F_i and G_i values

| Database Records | F_i | G_i |
|------------------|--------|--------|
| (0, 0, 1, 0) | Enc(0) | Enc(0) |
| (1, 0, 1, 1) | Enc(1) | Enc(1) |
| (1, 0, 0, 1) | Enc(0) | Enc(1) |
| (1, 0, 1, 1) | Enc(1) | Enc(2) |
| (1, 1, 0, 0) | Enc(0) | Enc(2) |

The resulting sequences (F'_i) would be similar as in Gahi's scheme,

$$\begin{aligned}
(F'_1) &= (\text{Enc}(0)) \\
(F'_2) &= (\text{Enc}(1), \text{Enc}(0)) \\
(F'_3) &= (\text{Enc}(0), \text{Enc}(0), \text{Enc}(0)) \\
(F'_4) &= (\text{Enc}(0), \text{Enc}(1), \text{Enc}(0), \text{Enc}(0)) \\
(F'_5) &= (\text{Enc}(0), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0)).
\end{aligned}$$

Therefore, the sequence (R') would be,

$$(R') = (\text{Enc}(R_2), \text{Enc}(R_3), \text{Enc}(0), \text{Enc}(0), \text{Enc}(0))$$

At this point, the sequence (R') will contain all the records that match our query but with trailing encryptions of zeros which we do not need. Hence, a second sum is calculated at the server side to determine the number of terms that are useful in the sequence:

$$n = \sum_r F_r.$$

Then n will be returned to the user and decrypted to obtain the number of records that match the search query. Hence, the sequence (R') can be truncated at the appropriate point and returned to the user for decryption.

It should be noted that the server will know the number of records that match the user's query. We believe that this information is not sufficient for the server to gain any additional information about the search query. Alternatively, we could return the whole sequence without truncation, keeping the number of matching records private from the server. However, the communication overhead will be increased significantly in this case, since the length of the sequence will be equal to the number of records in the database.

As promised previously, we now look at the special case where the record that is searched for is not contained anywhere in the database. In this case the value F_i will be something other than an encryption of 1 or 0. These garbage encrypted values will carry themselves into the rest of the protocol, resulting in Equation (8) with a nonsensical sequence. Hence, if the user receives a nonsensical sequence as the final result, it implies that the record that was searched is not contained in the database. As an alternative approach, we can compute $\prod_i (\text{Enc}(m) - \text{Enc}(R_i))$ prior to computing the F_i in Equation (5) and send it to the user to decrypt. If the result is zero then m is contained in the database, and if it is non-zero, m is not contained in the database and therefore the user can send a message to the server to abort the search.

5 Comparison of Our Scheme vs. Gahi's Scheme

Our scheme has the main advantage of having the potential to be used with more recent fully homomorphic encryption schemes rather than being restricted to the DGHV scheme. This gives the flexibility to use our method with block based encryption schemes such as Brakerski's [5], which reduces the number of encryption steps. For example, referring back to Equation (1), we can see that the I_t values are calculated by comparing the query with each record bit-wise. If there are m records in the database and each of them are encrypted using n bits, the number of operations that are required to calculate all the I_t values will be $\mathcal{O}(nm)$. In our Homomorphic Query Processing method, Equation (5) acts as the analogue of Equation (1). However, the encryptions are done block-wise in our scheme, and hence the number of operations it would take to calculate the F_i value in Equation (5) will be $\mathcal{O}(m)$. For Equation (2) in Gahi's method, the number of operations that should be performed to calculate all the partial sums will be $\mathcal{O}(nm^2)$, since there are $\mathcal{O}(m^2)$ multiplications and each multiplication should be done bit-wise; whereas the calculation of partial sums in our scheme (Equation (6)), the number of operations is reduced to $\mathcal{O}(m^2)$. Similarly, equations 3 and 4 in Gahi's method use $\mathcal{O}(nm)$ and $\mathcal{O}(nm^2)$ number of operations, respectively, but their counterparts in our scheme, (equations 7 and 8) have $\mathcal{O}(m)$ and $\mathcal{O}(m^2)$ operations, respectively. Thus, it can be seen that in each step of our scheme, the number of operations performed is reduced by a factor of n compared to Gahi's method.

References

1. K. Zickuhr, *Location Based Services* URL: <http://www.pewinternet.org/2013/09/12/location-based-services/> [accessed: 2016-01-16].
2. M. Kaku, *Physics of the Future*, Anchor Books, 2012.
3. Y. Gahi, M. Guennoun, and K. El-Khatib, *A Secure Database System using Homomorphic Encryption Schemes*, The Third International Conference on Advances in Databases, Knowledge, and Data Applications, 2011.
4. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, *Fully Homomorphic Encryption over the Integers*, Advances in Cryptology – EUROCRYPT 2010, **6110** 24-43, 2010.
5. Z. Brakerski, V. Vaikuntanathan, *Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages*, Advances in Cryptology - CRYPTO, **6841** 505-524, 2011.

6. R.L. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, **21** 120-126, 1978.
7. H. Hesse and C. Matthies, *Introduction to Homomorphic Encryption*, Cloud Security Mechanisms, December 2013.
8. C. Gentry, A fully homomorphic encryption scheme, PhD Thesis, Stanford University, 2009.
9. D. Boneh, C. Gentry, S. Halevi, F. Wang, D.J. Wu, *Private Database Queries Using Somewhat Homomorphic Encryption*, Applied Cryptography and Network Security, **7954** 102-118, 2013.
10. J.W. Bos, K. Lauter, J. Loftus, and M. Naehrig, *Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme* Cryptography and Coding – 14th IMA International Conference, Springer Lecture notes in computer science, **8308** 45–64, 2013.
11. B. Harrison, *On the Reducibility of Cyclotomic Polynomials over Finite Fields* The American Mathematical Monthly, **114** 813-818, 2007.
12. T. Dierks, Transport Layer Security (TLS) Protocol Version 1.2, <https://tools.ietf.org/html/rfc5246> August 2008.